

# Lab 2 - First steps with Spark

---




## Outline

---

1. Launching a Spark cluster on AWS
2. First steps with Spark

### Lab 2 - First steps with Spark

#### Outline

-  Spark cluster creation in AWS
-  First steps with Spark - Data importation
  -  Hands-on 1 - Data importation
-  Data frame basic manipulations
  -  Lazy evaluation
  -  Hands-on 2 - Data frame basic manipulations
-  Basic DataFrame column manipulation
  -  Hands-on 3 - Basic DataFrame column manipulation
-  Advance DataFrame column manipulation
  -  Array manipulation
    -  Hands-on 4 - Array manipulation
  -  User defined function
    -  Hands-on 5 - User defined function
-  Aggregation functions
  -  Hands-on 6 - Aggregation functions
-  Grouping functions
  -  Hands-on 7 - Grouping functions
-  Spark SQL
  -  Hands-on 8 - Spark SQL
-  Joins in Spark
  -  Hands-on 9 - Joins in Spark

## Spark cluster creation in AWS

---

First: **DO NOT FORGET TO TURN YOUR CLUSTER OFF A THE END OF THIS TUTORIAL!**

- ☐ Once connected to the management console, search for "EMR" (Elastic Map Reduce). It a platform as a service made to manage Hadoop cluster in AWS. You just have to choose the configuration of your cluster (how many machines ? How many CPU/Ram ? Which release for Spark ?) and AWS will create your cluster. Doing this all by yourself is time consuming and not a pleasant task. That's why cloud providers provide service like EMR.

☐ You should land on a page like this

Amazon EMR

EMR on EC2

Clusters

Blocs-notes

Git repositories

Configurations de la sécurité

Bloquer l'accès public

Sous-réseaux VPC

Événements

EMR on EKS

Virtual clusters

Aide

Nouveautés

## Bienvenue dans Amazon Elastic MapReduce

Amazon Elastic MapReduce (Amazon EMR) est un service Web qui permet aux commerces, aux chercheurs, aux analystes de données et aux développeurs de traiter de grandes quantités de données de manière simple et économique.

Apparemment, vous n'avez aucun cluster. Créez-en un maintenant :

[Créer un cluster](#)

### Fonctionnement d'Elastic MapReduce

Charger

Créer

Contrôle

Téléchargez les données et l'application de traitement sur S3.

Configurez et créez votre cluster en spécifiant les entrées et les sorties de données, la taille du cluster, les

Surveillez l'état et la progression de votre cluster. Récupérez la sortie dans S3.

### Informations supplémentaires

En savoir plus sur Elastic MapReduce :

[Présentation d'EMR](#)

[FAQ](#)

[Tarification](#)

En savoir plus sur l'utilisation d'Elastic MapReduce :

[Forum](#)

[Documentation](#)

[Manuel du développeur](#)

[Référence API](#)

[EMR sur GitHub](#)

[Portail d'aide](#)

Next time it should be this one

Amazon EMR

EMR on EC2

Clusters

Blocs-notes

Git repositories

Configurations de la sécurité

Bloquer l'accès public

Sous-réseaux VPC

Événements

EMR on EKS

Virtual clusters

Aide

Nouveautés

[Créer un cluster](#)

[Afficher les détails](#)

[Cloner](#)

[Réinitialiser](#)

Filter : Tous les clusters

4 clusters (tous chargés)

	Nom	ID	Statut	Heure de création (UTC+1)	Temps écoulé	Heures normales
<input type="checkbox"/>	<a href="#">Mon cluster</a>	j-1H72GTU5MWKU2	Résilié Demande utilisateur	23-03-2021 15:23 (UTC+1)	53 minutes	24
<input type="checkbox"/>	<a href="#">Mon cluster</a>	j-38AR1CNY33PPB	Résilié Demande utilisateur	23-03-2021 15:15 (UTC+1)	11 minutes	24
<input type="checkbox"/>	<a href="#">NotebookCluster</a>	j-39CG54L47FWG	Résilié Demande utilisateur	23-03-2021 14:52 (UTC+1)	13 minutes	8
<input type="checkbox"/>	<a href="#">Mon cluster</a>	j-2P3UZ7CHZ3BHC	Résilié Demande utilisateur	23-03-2021 14:44 (UTC+1)	9 minutes	24

In every cases click on [Créer un cluster](#)

☐ One the next page, select this configuration :

- ☐ Nom du cluster : A descriptive name, like Cluster\_First\_Step\_With\_Spark
- ☐ Journalisation : uncheck it. It's only to save in S3 the log of your cluster. It's useful to debug your application when you run un script and not a notebook.
- ☐ Mode de lancement : `cluster`. The difference between `cluster` et `Exécution d'étape` is that `cluster` create a cluster made for interactive processing, and `Exécution d'étape` create a cluster, run one or several script and stop it at the end. It's useful to run long processes not for data exploration.
- ☐ Libérée (Release en VO) : `emr-5.31.0`, others versions have some issues
- ☐ Type d'instance : <https://pickerwheel.com/pw?id=LkjX5> spin to know your instance type. Those instances are the ones you can access with a educate account.
- ☐ Nombre d'instance : `3`, but every number between 2 and 6 should be fine (max 4 for c5.xlarge et m4.2xlarge).

Here is a table with the hourly price of some instances just to give you an idea of the cost of an EMR cluster

Instance	Hourly price per instance
m4.xlarge	0.26
m5.xlarge	0.24\$
m5.2xlarge	0.48\$
c4.large	0.12\$
c5.xlarge	0.21\$

☐ Paire de clé EC2 : select **Sans paire de clé EC2** ok **Aucune paire de clé trouvée**

☐ Then click on **Créer un cluster**

Your page should look like this :

**Créer un cluster - Options rapides** [Accéder aux options avancées](#)

#### Configuration générale

Nom du cluster    
☐ Journalisation   
Mode de lancement ☒ Cluster ☐ Exécution d'étape

A descriptive name

#### Configuration des logiciels

Libère    
Applications   
☐ Core Hadoop: Hadoop 2.10.0, Hive 2.3.7, Hue 4.7.1, Mahout 0.13.0, Pig 0.17.0, and Tez 0.9.2   
☐ HBase: HBase 1.4.13, Hadoop 2.10.0, Hive 2.3.7, Hue 4.7.1, Phoenix 4.14.3, and ZooKeeper 3.4.14   
☐ Presto: Presto 0.238.3 with Hadoop 2.10.0 HDFS and Hive 2.3.7 Metastore   
☒ Spark: Spark 2.4.6 on Hadoop 2.10.0 YARN and Zeppelin 0.8.2   
☐ Utiliser AWS Glue Data Catalog pour les métadonnées de table

Cautious ! If you choose the wrong release you will have to create another cluster later

#### Configuration du matériel

Type d'instance  Le type d'instance sélectionné ajoute un volume EBS GP2 par défaut de 64 GiB par instance. [En savoir plus](#)   
Nombre d'instances  (1 nœud maître et 2 nœuds principaux)   
Cluster scaling ☐ scale cluster nodes based on workload

#### Sécurité et accès

Paire de clés EC2  [Apprenez à créer une paire de clés EC2.](#)   
Autorisations ☒ Par défaut ☐ Personnalisé   
Utilisez les rôles IAM par défaut. Si des rôles sont absents, ils seront créés automatiquement pour vous avec des stratégies gérées pour les mises à jour automatiques de stratégies.   
Rôle EMR [EMR\\_DefaultRole](#)   
Profil d'instance EC2 [EMR\\_EC2\\_DefaultRole](#)

[Annuler](#) [Créer un cluster](#)

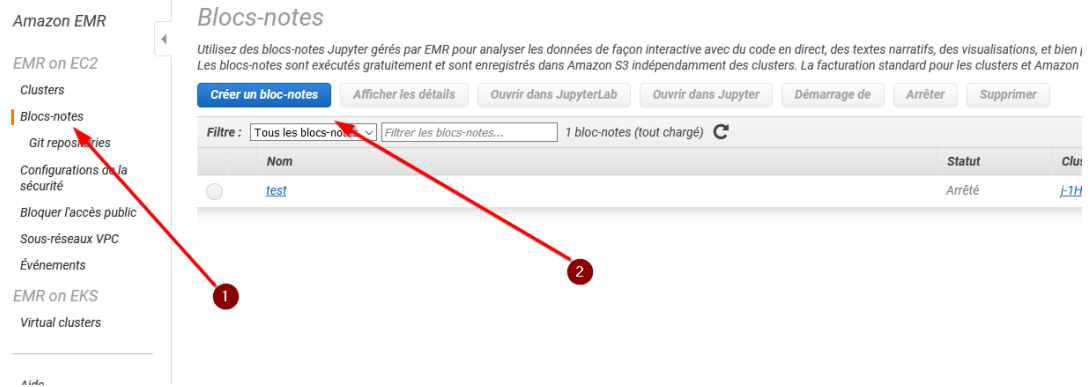
It's possible you get an error like this one :

Résilié avec des erreursThe requested instance type m5.xlarge is not supported in the requested availability zone. Learn more at [https://docs.aws.amazon.com/console/elasticmapreduce/ERROR\\_noinstancetype](https://docs.aws.amazon.com/console/elasticmapreduce/ERROR_noinstancetype)

The physical resources of AWS's datacenters are not unlimited, and AWS keeps some room for manoeuvre for top priority users. So sometimes we, low priority users, cannot use some resources. If this problem happens, just recreate a cluster with a less powerful machine, like m4.xlarge

⌚ The cluster creation takes time (around 5 min), please wait

- ☐ Once your cluster in "En attente" or "Stand by" (so after 5 min at least), go to **blocs-notes**, then click on **Créer un bloc-notes**



- ☐ Next you have to configure your notebook. It's pretty easy

- ☐ Give a descriptive name to your notebook
- ☐ Select your brand new cluster

And for this lab it's enough.

Your screen should look like this :

## Créer un bloc-notes

### Nommer et configurer votre bloc-notes

Nommez votre bloc-notes Jupyter géré par EMR, choisissez un cluster ou en créez-en un et personnalisez les options de configuration si vous le souhaitez. [En savoir plus](#)

**Nom du bloc-notes.\***

Les noms peuvent contenir uniquement des lettres (a-z), des chiffres (0-9), des traits d'union (-) ou des traits de soulignement (\_).

**Description**

256 caractères max.

**Cluster\*** ☒ Choisir un cluster existant

☐ Créer un cluster ⓘ

**Groupes de sécurité** ☒ Utiliser des groupes de sécurité par défaut ⓘ ☐ Choisir des groupes de sécurité

**Rôle de service AWS\***  ⓘ

**Emplacement du bloc-notes\*** Choose an S3 location where files for this notebook are saved.

☒ Use the default S3 location   
 s3://aws-emr-resources-523967347856-us-east-1/notebooks/

☐ Choose an existing S3 location in us-east-1

► **Référentiel Git** Lien vers un référentiel Git pour enregistrer votre bloc-notes dans un environnement de gestion des versions

► **Balises ⓘ**

\* Obligatoire

- ☐ The notebook creation should be fast. Once your notebook is ready click on **Ouvrir dans JupyterLab**. This will open JupiterLab. Create a notebook with a pyspark kernel and run the following line

```

1 #Spark session
2 spark
3
4 # Configuration
5 spark._jsc.hadoopConfiguration().set("fs.s3.useRequesterPaysHeader", "
  true")

```

If everything is ok you should get something like this after one or two minutes.

The screenshot shows a Jupyter Notebook titled "Lab 2 - First steps with Spark". It contains instructions for creating a Spark session and a table of Spark applications.

At this end of this lab :

- Save your notebook : right click on your notebook on the left panel > Download
- export your notebook : File > Export Notebook As ... > Export Notebook To HTML
- Turn your cluster off !

### Creating a Spark session

Write in a new cell :

- `spark` to check if everything is ok
- `spark._jsc.hadoopConfiguration().set("fs.s3.useRequesterPaysHeader", "true")` : to access to the data

```
[1]: #Spark session
spark

# Configuration
spark._jsc.hadoopConfiguration().set("fs.s3.useRequesterPaysHeader", "true")
```

Starting Spark application

ID	YARN Application ID	Kind	State	Spark UI	Driver log	Current session?
3	application_1618812226274_0004	pyspark	idle	<a href="#">Link</a>	<a href="#">Link</a>	✓

SparkSession available as 'spark'.

If not, just check if your cluster is `En attente`. If not, just wait, if so, ask for help.

**DO NOT FORGET TO TURN YOUR CLUSTER OFF AT THE END OF THIS TUTORIAL!**

## First steps with Spark - Data importation

Spark's main object class is the DataFrame, which is a distributed table. It is analogous to R's or Python (Pandas)'s data frames: one row represents an observation, one column represents a variable. But contrary to R or Python, Spark's DataFrames can be distributed over hundred of nodes.

Spark support multiple data formats, and multiple ways to load them.

- data format : csv, json, parquet (an open source column oriented format)
- can read archive files
- schema detection or user defined schema. For static data, like a json file, schema detection can be use with good results.

Spark has multiple syntaxes to import data. Some are simple with no customisation, others are more complexes but you can specify options.

The simplest syntaxes to load a json or a csv file are :

```

1 # JSON
2 json_df = spark.read.json([location of the file])
3 # CSV
4 csv_df = spark.read.csv([location of the file])
5


```

In the future, you may consult the [Data Source documentation](#) to have the complete description of Spark's reading abilities.

The data you will use in this lab are real data from the twitter [sampled stream API](#) and [filtered stream API](#). The tweets folder contains more than 50 files and more than 2 million tweets. The tweets was collected between the 14/04/2021 and the 18/04/2021. The total collection time was less than 10 hours.

---

## Hands-on 1 - Data importation

- Load the json file store here : `s3n://spark-lab-input-data-ensai20202021/tweets/tweets20210414-143312.jsonl.gz` and name you data frame `df_tweet`  
  
 This file is an a `JSONL` (JSON-line) format, which means that each line of it is a JSON object. A JSON object is just a Python dictionary or a JavaScript object and looks like this: `{ key1: value1, key2: ["array", "of", "many values"] }`. This file has been compressed into a `GZ` archive, hence the `.jsonl.gz` ending. Also this file is not magically appearing in your S3 storage. It is hosted on one of your teacher's bucket and has been made public, so that you can access it.
- It's possible to load multiple file in a unique DataFrame. It's useful when you have daily files and want to process them all. It's the same syntax as the previous one, just specify a folder. Like `s3n://spark-lab-input-data-ensai20202021/tweets/`. Name you DataFrame `df_tweet_big`

Now you have two DataFrames .

Remember that **Spark is lazy**, in the sense that it will avoid at all cost to perform unnecessary operations and wait to the last moment for performing only the duly requested computations. (Maybe you remember that R is lazy in that sense, but Spark is one degree more lazy than R.)

- Knowing that, do you think that when you run `spark.read.json()`, the data is actually migrated from S3 to the cluster ? If you want some data to be actually loaded, you can use the `show(n)` method (omitting `n` defaults to 20).
- Each time you will transform this DataFrame, the data will be transferred to your cluster. To avoid that, you need to cache your two DataFrame with the `cache()` method.

Sparks has very loose constraints on what you can actually store in a DataFrame column. The objects we just imported are actually quite messy.

- Use the `printSchema()` method to see the structure of one object.

**Spark's DataFrames are immutable:** there is no method to alter one specific value once one is created. This on purpose: mutations are famously hard to track, and Spark want to track them in order to avoid unnecessary computations. Suppressing mutations is actually the the best way to track changes.

Also, **DataFrames are distributed over the cluster:** they are split into blocks, ill-named **partitions**<sup>1</sup>, that are stored separately in the memory of the workers nodes. Since Spark is lazy evaluation, all reading and intermediary computation is only kept in memory as your data are being processed.

---

## Data frame basic manipulations

If DataFrames are immutable, they can however be **transformed** in other DataFrames, in the sense that a modified copy is returned. Such **transformations** include: filtering, sampling, dropping columns, selecting columns, adding new columns...

First, you can get information about the columns with:

```
1 df.columns      # get the column names
2 df.schema       # get the column names and their respective type
3 df.printSchema() # same, but human-readable
```

You can select columns with the `select()` method. It takes as argument a list of column name. For example :

```
1 df_with_less_columns = df\
2   .select("variable3", "variable_four", "variable-6")
3
4 # Yes, you do need the ugly \ at the end of the line,
5 # if you want to chain methods between lines in Python
```

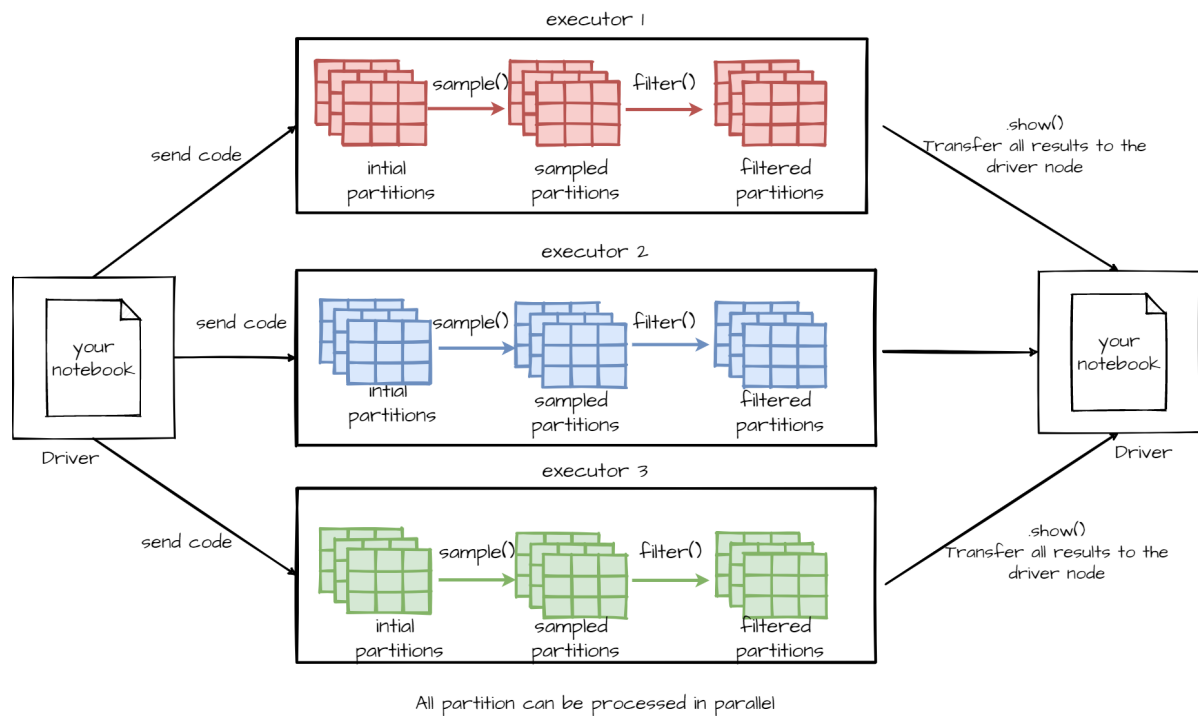
You can get nested columns easily with :

```
1 df.select("parentField.nestedField")
```

To filter data you could use the `filter()` method. It take as input an expression that gets evaluated for each observation and should return a boolean. Sampling is performed with the `sample()` method. For example :

```
1 df_with_less_rows = df\
2   .sample(fraction=0.001)\
3   .filter(df.variable1=="value")\
4   .show(10)
```

As said before your data are distributed over multiple nodes (executors) and data inside a node are split into partitions. Then each transformations will be run in parallel. They are called *narrow transformation* For example, to sample a DataFrame, Spark sample every partitions in parallel because sample all partition produce the sample DataFrame. For some transformations, like `groupBy()` it's impossible, and it's cannot be run in parallel.



## 🧠 Lazy evaluation

This is because Spark has what is known as **lazy evaluation**, in the sense that it will wait as much as it can before performing the actual computation. Said otherwise, when you run an instruction such as:

```
1 | tweet_author_hashtags = df_tweet_big.select("auteur", "hashtags")
```

... you are not executing anything! Rather, you are building an **execution plan**, to be realised later.

Spark is quite extreme in its laziness, since only a handful of methods called **actions**, by opposition to **transformations**, will trigger an execution. The most notable are:

1. `collect()`, explicitly asking Spark to fetch the resulting rows instead of to lazily wait for more instructions,
2. `take(n)`, asking for `n` first rows
3. `first()`, an alias for `take(1)`
4. `show()` and `show(n)`, human-friendly alternatives <sup>2</sup>
5. `count()`, asking for the numbers of rows
6. all the "write" methods (write on file, write to database), see [here](#) for the list

**This has advantages:** on huge data, you don't want to accidentally perform a computation that is not needed. Also, Spark can optimize each **stage** of the execution in regard to what comes next. For instance, filters will be executed as early as possible, since it diminishes the number of rows on which to perform later operations. On the contrary, joins are very computation-intense and will be executed as late as possible. The resulting **execution plan** consists in a **directed acyclic graph** (DAG) that contains the tree of all required actions for a specific computation, ordered in the most effective fashion.

**This has also drawbacks.** Since the computation is optimized for the end result, the intermediate stages are discarded by default. So if you need a DataFrame multiple times, you have to cache it in memory because if you don't Spark will recompute it every single time.



## Hands-on 2 - Data frame basic manipulations

- How many rows have your two DataFrame ?
- Sample `df_tweet_big` and keep only 10% of it. Create a new DataFrame named `df_tweet_sampled`. If computations take too long on the full DataFrame, use this one instead or add a sample transformation in your expression.
- Define a DataFrame `tweet_author_hashtags` with only the `auteur` and `hashtags` columns
- Print (few lines of) a DataFrame with only the `auteur`, `mentions`, and `urls` columns. (`mentions` and `urls` are both nested columns in `entities`.)
- Filter your first DataFrame and keep only tweets with more than 1 like. Give a name for this new, transformed DataFrame. Print (few lines of) it.

## Basic DataFrame column manipulation

You can add/update/rename column of a DataFrame with spark :

- Drop : `df.drop(columnName : str )`
- Rename : `df.withColumnRenamed(oldName : str, newName : str)`
- Add/update : `df.withColumn(columnName : str, columnExpression)`

For example

```
1 tweet_df_with_like_rt_ratio = tweet_df\  
2   .withColumn(           # computes new variable  
3     "like_rt_ratio", # like_rt_ratio "OVERCONFIDENCE"  
4     (tweet_df.like_count / flights.retweet_count  
5     )  
6
```

See [here](#) for the list of all functions available in an expression.

## Hands-on 3 - Basic DataFrame column manipulation

- Define a DataFrame with a column names `interaction_count`. This column is the sum of `like_count`, `reply_count` and `retweet_count`.
- Update the DataFrame you imported at the beginning of this lab and drop the `other` column

## Advance DataFrame column manipulation

### Array manipulation

Some columns often contain arrays (lists) of values instead of just one value. This may seem surprising but this actually quite natural. For instance, you may create an array of words from a text, or generate a list of random numbers for each observation, etc.

You may **create array of values** with:

- `split(text : string, delimiter : string)`, turning a text into an array of strings

You may **use array of values** with:

- `size(array : Array)`, getting the number of elements

- `array_contains(inputArray : Array, value : any)`, checking if some value appears
- `explode(array : Array)`, unnesting an array and duplicating other values. For instance it if use `explode()` over the `hashtags` value of this DataFrame:

Auteur	Contenu	Hashtags
Bob	I love #Spark and #bigdata	[Spark, bigdata]
Alice	Just finished #MHrise, best MH ever	[MHrise]

I will get :

Auteur	Contenu	Hashtags	Hashtag
Bob	I love #Spark and #bigdata	[Spark, bigdata]	Spark
Bob	I love #Spark and #bigdata	[Spark, bigdata]	bigdata
Alice	Just finished #MHrise, best MH ever	[MHrise]	MHrise

All this function must be imported first :

```
1 | from pyspark.sql.functions import split, explode, size, array_contains
```

Do not forget, to create a new column, you should use `withColumn()`. For example :

```
1 | df.withColumn("new column", explode("array"))
```

## Hands-on 4 - Array manipulation

- Keep all the tweets with hashtags and for each remaining line, split the hashtag text into an array of hashtags
- Create a new column with the number of words of the `contenu` column. (Use `split()` + `size()`)
- Count how many tweet contain the `COVID19` hashtag (use the `count()` action)

## User defined function

For more very specific column manipulation you will need Spark's `udf()` function (*User Defined Function*). It can be useful if you Spark does not provide a feature you want. But Spark is a popular and active project, so before coding an udf, go check the documentation. For instance for natural language processing, Spark already has some [functions](#). Last things, python udf can lead to performance issues (see <https://stackoverflow.com/a/38297050>) and learning a little bit of scala or java can be a good idea.

For example :

```

1 # !!!! DOES NOT WORK !!!!
2 def to_lower_case(string):
3     return string.lower()
4
5 df.withColumn("tweet_lower_case", to_lower_case(df.contenu))

```

will just crash. Keep in mind that Spark is a distributed system, and that Python is only installed on the central node, as a convenience to let you execute instructions on the executor nodes. But by default, pure Python functions can only be executed where Python is installed! We need `udf()` to enable Spark to send Python instructions to the worker nodes.

Let us see how it is done :

```

1 # imports
2 from pyspark.sql.functions import udf
3 from pyspark.sql.functions import explode
4 from pyspark.sql.types import StringType
5
6 # pure python functions
7 def to_lower_case(string):
8     return string.lower()
9
10 # user definid function
11 to_lower_case_udf = udf(
12     lambda x: to_lower_case(x), StringType()
13 ) #we use a lambda function to create the udf.
14
15 # df manipulation
16 df_tweet\
17     .select("auteur", "hashtags")\
18     .filter("size(hashtags)!=0")\
19     .withColumn("hashtag", explode("hashtags"))\
20     .withColumn("hashtag", to_lower_case_udf("hashtag")).show(10)

```

## Hands-on 5 - User defined function

- Create an user defined function that counts how many words a tweet contains. (your function will return an `IntegerType` and not a `StringType`)

## Aggregation functions

Spark offer a variety of aggregation functions :

- `count(column : string)` will count every not null value of the specify column. You cant use `count(1)` of `count("*")` to count every line (even row with only null values)
- `countDisctinct(column : string)` and `approx_count_distinct(column : string, percent_error: float)`. If the exact number is irrelevant, `approx_count_distinct()` should be preferred.

Counting distinct elements cannot be done in parallel, and need a lot data transfer. But if you only need an approximation, there is a algorithm, named hyper-log-log (more info [here](#)) that can be parallelized.

```

1 from pyspark.sql.functions import count, countDistinct,
  approx_count_distinct
2
3 df.select(count("col1")).show()
4 df.select(countDistinct("col1")).show()
5 df.select(approx_count_distinct("col1"), 0.1).show()

```

- You have access to all other common functions `min()`, `max()`, `first()`, `last()`, `sum()`, `sumDistinct()`, `avg()` etc (you should import them first `from pyspark.sql.functions import min, max, avg, first, last, sum, sumDistinct`)

## Hands-on 6 - Aggregation functions

- What are the min, max, average of `interaction_count`
- How many tweets have hashtags ? Distinct hashtags ? Try the approximative count with 0.1 and 0.01 as maximum estimation error allowed.

## Grouping functions

Like SQL you can group row by a criteria with Spark. Just use the `groupBy(column : string)` method. Then you can compute some aggregation over those groups.

```

1 df.groupBy("col1").agg(
2     count("col2").alias("quantity") # alias is use to specify the name of the
  new column
3 ).show()

```

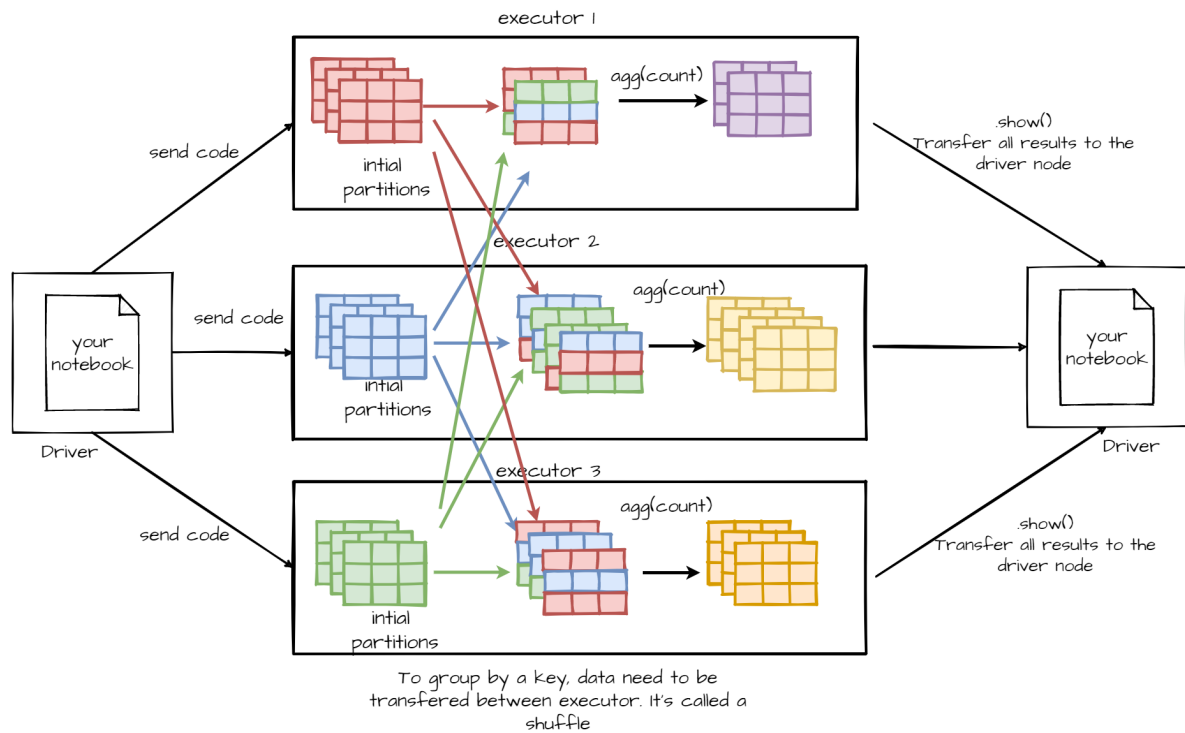
The `agg()` method can take multiples argument to compute multiple aggregation at once.

```

1 df.groupBy("col1").agg(
2     count("col2").alias("quantity"), min("col2").alias("min"),
  avg("col3").alias("avg3") ).show()

```

Aggregation and grouping transformations work differently than the previous method like `filter()`, `select()`, `withColumn()` etc. Those transformations cannot be run over each partitions in parallel, and need to transfer data between partitions and executors. They are called "wide transformations"



## Hands-on 7 - Grouping functions

- Compute a dataframe with the min, max and average retweet of each `auteur`. Then order it by the max number of retweet in descending order by . To do that you can use the following syntax

```
1 from pyspark.sql.functions import desc
2 df.orderBy(desc("col"))
```

## Spark SQL

Spark understand SQL statement. It's not a hack nor a workaround to use SQL in Spark, it's one of the more powerful features in Spark. To use SQL, you need:

1. Register a view pointing to your DataFrame. In SQL statement you will refer to your DataFrame with its view name

```
1 my_df.createOrReplaceTempView(viewName : str)
```

2. Use the sql function

```
1 spark.sql("""
2 You sql statment
3 """)
```

You could manipulate every registered DataFrame by their view name with plain SQL. For instance

```

1 df_tweet.createOrReplaceTempView("small_tweet_df")
2 spark.sql("""
3 SELECT *
4 FROM small_tweet_df
5 """)

```

In fact you can do most of this tutorial without any knowledge in PySpark nor Spark. Lot of things can be done in Spark only by only knowing SQL and how to use it in Spark.

## Hands-on 8 - Spark SQL

- How many tweets have hashtags ? Distinct hashtags ?
- Compute a dataframe with the min, max and average retweet of each `auteur` using Spark SQL

## Joins in Spark

Like a SQL, Spark can join two dataset by comparing the value of one or more keys using joins. Joins are by nature wide transformation, so data will be transferred between executors and will take time. But Spark will distinct two cases :

- Big table to big table join
- Big table to small table join
- (The case small table to small table is irrelevant for big data)

And optimize the join according to the actual case. (For more information Spark: The Definitive Guide pages 148 - 151)

Doing a join is pretty easy. You need :

- At least two DataFrames (obviously) with columns with the same keys
- a join expression
- the join transformation

For instance :

```

1 # Creation of 3 small DF
2 person=spark.createDataFrame([
3     (0,"Bill Chambers",0,[100])
4     ,(1,"Matei Zaharia",1,[500,250,100])
5     ,(2,"Michael Armbrust",1,[250,100])])\
6     .toDF("id","name","graduate_program","spark_status")
7 graduateProgram=spark.createDataFrame([
8     (0,"Masters","School of Information","UC Berkeley")
9     ,(2,"Masters","EECS","UC Berkeley"),(1,"Ph.D.","EECS","UC Berkeley")])\
10    .toDF("id","degree","department","school")
11 sparkStatus=spark.createDataFrame([
12     (500,"Vice President")
13     ,(250,"PMC Member")
14     ,(100,"Contributor")])\
15    .toDF("id","status")
16
17 # A join expression
18 joinExpression=person["graduate_program"]==graduateProgram['id']
19
20 # The join transformation in action

```

By default Spark compute inner joins, but you can pass a third argument to the join transformation with its type. You can do :

- Inner joins : by default or with the "inner" argument
- Outer joins : "outer" argument
- Left / right outer joins : "left\_outer", "right\_outer" argument
- Left semi join : it's more a filter than a join. It only keep the row in the left DataFrame that have a match in the right DataFrame. Use the "left\_semi" argument
- Left anti join : The opposite of the previous one. Use the "left\_anti" argument
- You can do cross joins to, but it's a very bad idea to do so, so please don't !

## Hands-on 9 - Joins in Spark

- Import the files stored in `s3n://spark-lab-input-data-ensai20202021/users/` in a DataFrame and its informations to your DataFrame. Filter your new DataFrame to only keep verified user (`verified == True`) and group by user and get the most active user of your DataFrame.

This time you will use a user defined schema to load your data, and pass it to the `json()` method with the `schema` parameter. Here is the schema of the user data :

```
1 from pyspark.sql.types import StructType, StructField, StringType,
  IntegerType, ArrayType, TimestampType, BooleanType
2 schema = StructType([ \
3     StructField("created_at", TimestampType(), True), \
4     StructField("id", StringType(), True), \
5     StructField("name", StringType(), True), \
6     StructField("username", StringType(), True), \
7     StructField("withheld", StructType([
8         StructField("country_codes", ArrayType(StringType(), True)), \
9         StructField("scope", StringType(), True) \
10    ])),
11     StructField("verified", BooleanType(), True)
12 ])
```

- Filter the data to only keep tweets create by user create after 2019. You can use `to_date(lit("2019-01-01"))` and a `filter()` transformation to do so.

**DO NOT FORGET TO TURN YOUR CLUSTER OFF!**

**DO NOT FORGET TO TURN YOUR CLUSTER OFF!**

**DO NOT FORGET TO TURN YOUR CLUSTER OFF!**

**DO NOT FORGET TO TURN YOUR CLUSTER OFF!**

**DO NOT FORGET TO TURN YOUR CLUSTER OFF!**

**DO NOT FORGET TO TURN YOUR CLUSTER OFF!**

**DO NOT FORGET TO TURN YOUR CLUSTER OFF!**

**DO NOT FORGET TO TURN YOUR CLUSTER OFF!**

replication is enforced, intersections between blocks are not necessarily empty. However, the union of all the blocks do produce the full original set. [↩](#)

2. `first()` is exactly `take(1)` `(ref)` and show prints the result instead of returning it as a list of rows `(ref)` [↩](#)